Chapter 7

Development methodologies for GPU and cluster of GPUs

Sylvain Contassot-Vivier

Université Lorraine, Loria UMR 7503 & AlGorille INRIA Project Team, Nancy, France.

Stephane Vialle

SUPELEC, UMI GT-CNRS 2958 & AlGorille INRIA Project Team, Metz, France.

Jens Gustedt

INRIA Nancy – Grand Est, AlGorille INRIA Project Team, Strasbourg, France.

7.1 Introduction

This chapter proposes to draw several development methodologies to obtain efficient codes in classical scientific applications. Those methodologies are based on the feedback from several research works involving GPUs, either alone in a single machine or in a cluster of machines. Indeed, our past collaborations with industries have allowed us to point out that in their economical context, they can adopt a parallel technology only if its implementation and maintenance costs are small according to the potential benefits (performance, accuracy,...). So, in such contexts, GPU programming is still regarded with some distance according to its specific field of applicability (SIMD/SIMT model) and its still higher programming complexity and maintenance. In the academic domain, things are a bit different but studies for efficiently integrating GPU computations in multi-core clusters with maximal overlapping of computations with communications and/or other computations, are still rare.

For these reasons, the major aim of that chapter is to propose as simple as possible general programming patterns that can be followed or adapted in practical implementations of parallel scientific applications. Also, we propose in a third part, a prospect analysis together with a particular programming tool that is intended to ease multi-core GPU cluster programming.

7.2 General scheme of synchronous code with computation/communication overlapping in GPU clusters

7.2.1 Synchronous parallel algorithms on GPU clusters

Considered parallel algorithms and implementations

This section focusses on synchronous parallel algorithms implemented with overlapping computations and communications. Parallel synchronous algorithms are easier to implement, debug and maintain than asynchronous ones, see Section 7.3. Usually, they follow a BSP-like parallel scheme, alternating local computation steps and communication steps, see [19]. Their execution is usually deterministic, excepted for stochastic algorithms that contain random number generations. Even in this case, their execution can be controlled during debug steps, allowing to track and to fix bugs quickly.

However, depending on the properties of the algorithm, it is sometimes possible to overlap computations and communications. If processes exchange data that is not needed for the computation that is following immediately, it is possible to implement such an overlap. We have investigated the efficiency of this approach in previous works [14, 21], using standard parallel programming tools to achieve the implementation.

The normalized and well known Message Passing Interface (MPI) includes some asynchronous point-to-point communication routines, that should allow to implement some communication/computation overlap. However, current MPI implementations do not achieve that goal efficiently; effective overlapping with MPI requires a group of dedicated threads (in our case implemented with OpenMP) for the basic synchronous communications while another group of threads executes computations in parallel. Nevertheless, communication and computation are not completely independent on modern multicore architectures: they use shared hardware components such as the interconnection bus and the RAM. Therefore that approach only saved up to 20% of the expected time on such a platform. This picture changes on clusters equipped with GPU. They effectively allow independence of computations on the GPU and communication on the mainboard (CPU, interconnection bus, RAM, network card). We saved up to 100% of the expected time on our GPU cluster, as we will expose in the next section.

Specific interests in GPU clusters

In a computing node, a GPU is a kind of scientific coprocessor usually located on an auxiliary board, with its own memory. So, when data have been transferred from the CPU memory to the GPU memory, then GPU computations can be achieved on the GPU board, totally in parallel of any CPU activities (like internode cluster communications). The CPU and the GPU access their respective memories and do not interfere, so they can achieve a very good overlap of their activities (better than two CPU cores).

But using a GPU on a computing node requires to transfer data from the CPU to the GPU memory, and to transfer the computation results back from the GPU to the CPU. Transfer times are not excessive, but depending on the application they still can be significant compared to the GPU computation times. So, sometimes it can be interesting to overlap the internode cluster communications with both the CPU/GPU data transfers and the GPU computations. We can identify four main parallel programming schemes on a GPU cluster:

1. parallelizing only 'internode CPU communications' with 'GPU computations', and achieving CPU/GPU data transfers before and after this parallel step,
2. parallelizing 'internode CPU communications' with a '(sequential) sequence of CPU/GPU data transfers and GPU computations',
3. parallelizing 'internode CPU communications' with a 'streamed sequence of CPU/GPU data transfers and GPU computations',
4. parallelizing 'internode CPU communications' with 'CPU/GPU data

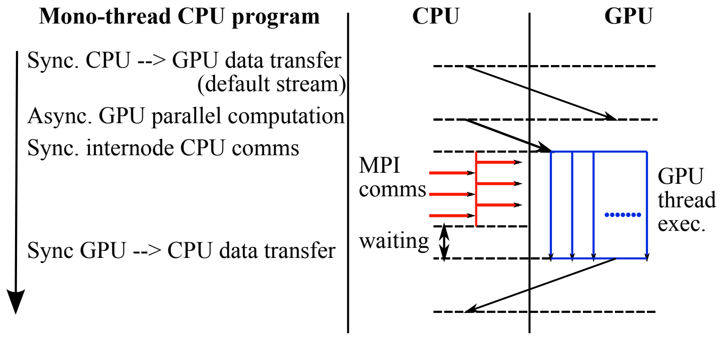
Mono-thread CPU program
CPU
GPU
Sync. CPU --> GPU data transfer (default stream)
Async. GPU parallel computation
Sync. internode CPU comms
MPI comms
GPU thread exec.
waiting
Sync GPU --> CPU data transfer

FIGURE 7.1: Native overlap of internode CPU communications with GPU computations.

transfers' and with 'GPU computations', interleaving computation-communication iterations.

7.2.2 Native overlap of CPU communications and GPU computations

Using CUDA, GPU kernel executions are non-blocking, and GPU/CPU data transfers are blocking or non-blocking operations. All GPU kernel executions and CPU/GPU data transfers are associated to "streams", and all operations on a same stream are serialized. When transferring data from the CPU to the GPU, then running GPU computations and finally transferring results from the GPU to the CPU, there is a natural synchronization and serialization if these operations are achieved on the same stream. GPU developers can choose to use one (default) or several streams. In this first scheme of overlapping, we consider parallel codes using only one GPU stream.

"Non-blocking GPU kernel execution" means a CPU routine runs a parallel execution of a GPU computing kernel, and the CPU routine continues its execution (on the CPU) while the GPU kernel is running (on the GPU). Then the CPU routine can initiate some communications with some others CPU, and so it automatically overlaps the internode CPU communications with the GPU computations (see Figure 7.1). This overlapping is natural when programming with CUDA and MPI: it is easy to deploy, but does not overlap the CPU/GPU data transfers.

Listing 7.1: Generic scheme implicitly overlapping MPI communications with CUDA GPU computations

```
   // Input data and result variables and arrays (example with
   // float datatype, 1D input arrays, and scalar results)
   float *cpuInputTabAdr, *gpuInputTabAdr;
   float *cpuResTabAdr, *gpuResAdr;
 5
   // CPU and GPU array allocations
   cpuInputTabAdr = malloc(sizeof(float)*N);
   cudaMalloc(&gpuInputTabAdr, sizeof(float)*N);
   cpuResTabAdr = malloc(sizeof(float)*NbIter);
10 cudaMalloc(&gpuResAdr, sizeof(float));

   // Definition of the Grid of blocks of GPU threads
   dim3 Dg, Db;
   Dg.x = ...
15 ...

   // Indexes of source and destination MPI processes
   int dest = ...
   int src = ...
20
   // Computation loop (using the GPU)
   for (int i = 0; i < NbIter; i++) {
     cudaMemcpy(gpuInputTabAdr, cpuInputTabAdr,   // Data transfer:
                sizeof(float)*N,                  // CPU --> GPU (sync. op
                    )
25              cudaMemcpyHostToDevice);
     gpuKernel_k1<<<Dg,Db>>>();                   // GPU comp. (async. op)
     MPI_Sendrecv_replace(cpuInputTabAdr,         // MPI comms. (sync. op)
                          N,MPI_FLOAT,
                          dest, 0, src, 0, ...);
30   // IF there is (now) a result to transfer from the GPU to the CPU:
     cudaMemcpy(cpuResTabAdr + i, gpuResAdr,      // Data transfer:
                sizeof(float),                    // GPU --> CPU (sync. op
                    )
                cudaMemcpyDeviceToHost);
   }
35 ...
```

Listing 7.1 introduces the generic code of a MPI+CUDA implementation, natively and implicitly overlapping MPI communications with CUDA GPU computations. Some input data and output results arrays and variables are declared and allocated from line 1 up to 10, and a computation loop is implemented from line 21 up to 34. At each iteration:

- `cudaMemcpy` on line 23 transfers data from the CPU memory to the GPU memory. This is a basic and synchronous data transfer.

- `gpuKernel_k1<<<Dg,Db>>>` on line 26 starts GPU computation (running a GPU kernel on the grid of blocks of threads defined at line 12 to 15). This is a standard GPU kernel run, it is an asynchronous operation. The CPU can continue to run its code.

- `MPI_Sendrecv_replace` on line 27 achieves some blocking internode communications, overlapping GPU computations started just before.

- If needed, `cudaMemcpy` on line 31 transfers the iteration result from one variable in the GPU memory at one array index in the CPU memory

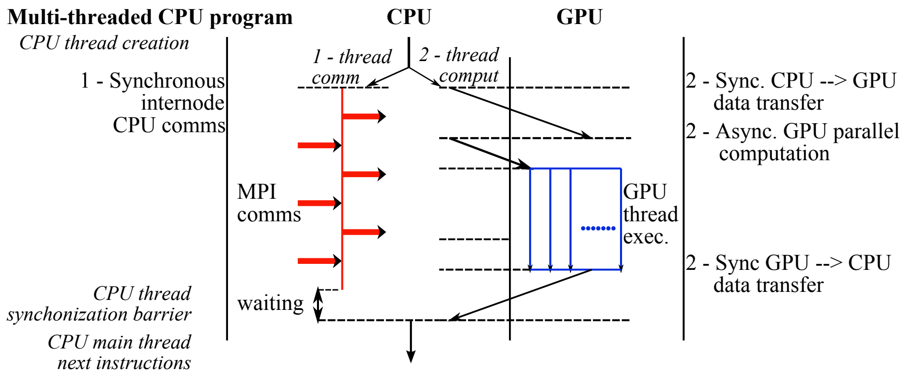
Multi-threaded CPU program
CPU
GPU
CPU thread creation
1 - thread comm
2 - thread comput
1 - Synchronous internode CPU comms
2 - Sync. CPU --> GPU data transfer
2 - Async. GPU parallel computation
MPI comms
GPU thread exec.
2 - Sync GPU --> CPU data transfer
CPU thread synchonization barrier
waiting
CPU main thread next instructions

FIGURE 7.2: Overlap of internode CPU communications with a sequence of CPU/GPU data transfers and GPU computations.

(in this example the CPU collects all iteration results in an array). This operation is started after the end of the MPI communication (previous instruction) and after the end of the GPU kernel execution. CUDA insures an implicit synchronization of all operations involving the same GPU stream, like the default stream in this example. Result transfer has to wait the GPU kernel execution is finished. If there is no result transfer implemented, the next operation on the GPU will wait until the GPU kernel execution will be ended.

This implementation is the easiest one involving the GPU. It achieves an implicit overlap of internode communications and GPU computations, no explicit multithreading is required on the CPU. However, CPU/GPU data transfers are achieved serially and not overlapped.

7.2.3 Overlapping with sequences of transfers and computations

Overlapping with a sequential GPU sequence

When CPU/GPU data transfers are not negligible compared to GPU computations, it can be interesting to overlap internode CPU computations with a *GPU sequence* including CPU/GPU data transfers and GPU computations (see Figure 7.2). Algorithmic issues of this approach are basic, but their implementation require explicit CPU multithreading and synchronization, and CPU data buffer duplication. We need to implement two threads, one starting and achieving MPI communications, and the other running the *GPU sequence*. OpenMP allows an easy and portable implementation of this overlapping strategy. However, it remains more complex to develop and to maintain than the previous strategy (overlapping only internode CPU communications and GPU computations), and should be adopted only when CPU/GPU data transfer times are not negligible.

Listing 7.2: Generic scheme explicitly overlapping MPI communications with sequences of CUDA CPU/GPU transfers and CUDA GPU computations

```
// Input data and result variables and arrays (example with
// float datatype, 1D input arrays, and scalar results)
float *cpuInputTabAdrCurrent, *cpuInputTabAdrFuture, *gpuInputTabAdr;
float *cpuResTabAdr, *gpuResAdr;

// CPU and GPU array allocations
cpuInputTabAdrCurrent = malloc(sizeof(float)*N);
cpuInputTabAdrFuture = malloc(sizeof(float)*N);
cudaMalloc(&gpuInputTabAdr,sizeof(float)*N);
cpuResTabAdr = malloc(sizeof(float)*NbIter);
cudaMalloc(&gpuResAdr,sizeof(float));

// Definition of the Grid of blocks of GPU threads
dim3 Dg, Db;
Dg.x = ...
...

// Indexes of source and destination MPI processes
int dest = ...
int src = ...

// Set the number of OpenMP threads (to create) to 2
omp_set_num_threads(2);
// Create threads and start the parallel OpenMP region
#pragma omp parallel
{
  // Buffer pointers (thread local variables)
  float *current = cpuInputTabAdrCurrent;
  float *future = cpuInputTabAdrFuture;
  float *tmp;

  // Computation loop (using the GPU)
  for (int i = 0; i < NbIter; i++) {

    // - Thread 0: achieves MPI communications
    if (omp_get_thread_num() == 0) {
      MPI_Sendrecv(current,                    // MPI comms. (sync. op)
                   N, MPI_FLOAT, dest, 0,
                   future,
                   N, MPI_FLOAT, dest, 0, ...);

    // - Thread 1: achieves the GPU sequence (GPU computations and
    //             CPU/GPU data transfers)
    } else if (omp_get_thread_num() == 1) {
      cudaMemcpy(gpuInputTabAdr, current,      // Data transfer:
                 sizeof(float)*N,              // CPU --> GPU (sync. op
                     )
                 cudaMemcpyHostToDevice);
      gpuKernel_k1<<<Dg,Db>>>();               // GPU comp. (async. op)
      // IF there is (now) a result to transfer from the GPU to the
          CPU:
      cudaMemcpy(cpuResTabAdr + i, gpuResAdr, // Data transfer:
                 sizeof(float),                // GPU --> CPU (sync. op
                     )
                 cudaMemcpyDeviceToHost);
    }

    // - Wait for both threads have achieved their iteration tasks
    #pragma omp barrier
    // - Each thread permute its local buffer pointers
```

```
      tmp = current;
      current = future;
      future = tmp;
    } // End of computation loop
} // End of OpenMP parallel region
...
```

Listing 7.2 introduces the generic code of a MPI+OpenMP+CUDA implementation, explicitly overlapping MPI communications with *GPU sequences*. Lines 25–62 implement the OpenMP parallel region, around the computation loop (lines 33–61). For performances it is important to create and destroy threads only one time (not at each iteration): the parallel region has to surround the computation loop. Lines 1–11 consist in declaration and allocation of input data arrays and result arrays and variables, like in previous algorithm (Listing 7.1). However, we implement two input data buffers on the CPU (current and future version). As we aim to overlap internode MPI communications and GPU sequence, including CPU to GPU data transfer of current input data array, we need to store the received new input data array in a separate buffer. Then, the current input data array will be safely read on the CPU and copied into the GPU memory.

The thread creations are easily achieved with one OpenMP directive (line 25). Then each thread defines and initializes *its* local buffer pointers, and enters *its* computing loop (lines 27–33). Inside the computing loop, a test on the thread number allows to run a different code in each thread. Lines 37–40 implement the MPI synchronous communication run by thread number 0. Lines 45–52 implement the GPU sequence run by thread 1: CPU to GPU data transfer, GPU computation and GPU to CPU result transfer (if needed). Details of the three operations of this sequence have not changed compared to the previous overlapping strategy.

At the end of Listing 7.2, an OpenMP synchronization barrier on line 56 allows to wait OpenMP threads have achieved MPI communications and GPU sequence, and do not need to access the current input data buffer. Then, each thread permute its local buffer pointers (lines 58–60), and is ready to enter the next iteration, processing the new current input array.

Overlapping with a streamed GPU sequence

Depending on the algorithm implemented, it is sometimes possible to split the GPU computation into several parts processing distinct data. Then, we can speedup the *GPU sequence* using several CUDA *streams*. The goal is to overlap CPU/GPU data transfers with GPU computations inside the *GPU sequence*. Compared to the previous overlapping strategy, we have to split the initial data transfer in a set of n asynchronous and smaller data transfers, and to split the initial GPU kernel call in a set of n calls to the same GPU kernel. Usually, these smaller calls are deployed with less GPU threads (i.e. associated to a smaller grid of blocks of threads). Then, the first GPU computations can start as soon as the first data transfer has been achieved, and next transfers can be done in parallel of next GPU computations (see Figure 7.3).

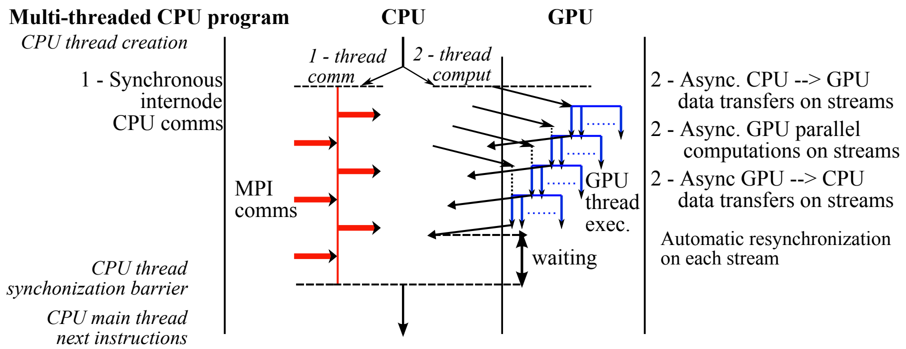
Multi-threaded CPU program
CPU
GPU
CPU thread creation
1 - thread comm
2 - thread comput
1 - Synchronous internode CPU comms
MPI comms
GPU thread exec.
waiting
2 - Async. CPU --> GPU data transfers on streams
2 - Async. GPU parallel computations on streams
2 - Async GPU --> CPU data transfers on streams
Automatic resynchronization on each stream
CPU thread synchonization barrier
CPU main thread next instructions

FIGURE 7.3: Overlap of internode CPU communications with a streamed sequence of CPU/GPU data transfers and GPU computations.

NVIDIA advises to start all asynchronous CUDA data transfers, and then to call all CUDA kernel executions, using up to 16 streams [17]. Then, CUDA driver and runtime optimize the global execution of these operations. So, we cumulate two overlapping mechanisms. The former is controlled by CPU multithreading, and overlap MPI communications and the *streamed GPU sequence*. The latter is controlled by CUDA programming, and overlap CPU/GPU data transfers and GPU computations. Again, OpenMP allows to easily implement the CPU multithreading, and to wait for the end of both CPU threads before to execute the next instructions of the code.

Listing 7.3: Generic scheme explicitly overlapping MPI communications with streamed sequences of CUDA CPU/GPU transfers and CUDA GPU computations

```
// Input data and result variables and arrays (example with
// float datatype, 1D input arrays, and scalar results)
float *cpuInputTabAdrCurrent, *cpuInputTabAdrFuture, *gpuInputTabAdr;
float *cpuResTabAdr, *gpuResAdr;
// CPU and GPU array allocations (allocates page-locked CPU memory)
cudaHostAlloc(&cpuInputTabAdrCurrent,sizeof(float)*N,
     cudaHostAllocDefault);
cudaHostAlloc(&cpuInputTabAdrFuture,sizeof(float)*N,
     cudaHostAllocDefault);
cudaMalloc(&gpuInputTabAdr,sizeof(float)*N);
cpuResTabAdr = malloc(sizeof(float)*NbIter);
cudaMalloc(&gpuResAdr,sizeof(float));
// Stream declaration and creation
cudaStream_t TabS[NbS];
for(int s = 0; s < NbS; s++)
  cudaStreamCreate(&TabS[s]);
// Definition of the Grid of blocks of GPU threads
...
// Set the number of OpenMP threads (to create) to 2
omp_set_num_threads(2);
// Create threads and start the parallel OpenMP region
#pragma omp parallel
{
  // Buffer pointers (thread local variables)
  float *current = cpuInputTabAdrCurrent;
  float *future = cpuInputTabAdrFuture;
25    float *tmp;
      // Stride of data processed per stream
      int stride = N/NbS;
      // Computation loop (using the GPU)
      for (int i = 0; i < NbIter; i++) {
30      // - Thread 0: achieves MPI communications
        if (omp_get_thread_num() == 0) {
          MPI_Sendrecv(current,               // MPI comms. (sync. op)
                       N, MPI_FLOAT, dest, 0,
                       future,
35                     N, MPI_FLOAT, dest, 0, ...);
        // - Thread 1: achieves the streamed GPU sequence (GPU
            computations
        //             and CPU/GPU data transfers)
        } else if (omp_get_thread_num() == 1) {
          for (int s = 0; s < NbS; s++) {     // Start all data transfers
              :
40          cudaMemcpyAsync(gpuInputTabAdr + s*stride, // CPU —> GPU
                            current + s*stride,        // (async. ops)
                            sizeof(float)*stride,
                            cudaMemcpyHostToDevice,
                            TabS[s]);
45        }
          for (int s = 0; s < NbS; s++) {   // Start all GPU comps. (async
              .)
            gpuKernel_k1<<<Dg, Db, 0, TabS[s]>>>(gpuInputTabAdr + s*stride
                );
          }
          cudaThreadSynchronize();            // Wait all threads are
              ended
50        // IF there is (now) a result to transfer from the GPU to the
             CPU:
          cudaMemcpy(cpuResTabAdr,            // Data transfers:
                     gpuResAdr,               // GPU —> CPU (sync. op)
                     sizeof(float),
                     cudaMemcpyDeviceToHost);
55      }
        // - Wait for both threads have achieved their iteration tasks
        #pragma omp barrier
        // - Each thread permute its local buffer pointers
        tmp = current; current = future; future = tmp;
60    } // End of computation loop
    } // End of OpenMP parallel region
    ...
    // Destroy the streams
    for(int s = 0; s < NbS; s++)
65    cudaStreamDestroy(TabS[s]);
    ...
```

Listing 7.3 introduces the generic MPI+OpenMP+CUDA code explicitly overlapping MPI communications with *streamed GPU sequences*. Efficient usage of CUDA *streams* requires to execute asynchronous CPU/GPU data transfers, that needs to read page-locked data in CPU memory. So, CPU memory allocations on lines 6 and 7 are implemented with `cudaHostAlloc` instead of the basic `malloc` function. Then, *NbS streams* are created on lines 12–14. Usually we create 16 streams: the maximum number supported by CUDA.

An OpenMP parallel region including two threads is implemented on lines 17–61 of Listing 7.3, similarly to the previous algorithm (see Listing 7.2). Code of thread 0 achieving MPI communication is unchanged, but code of thread 1 is now using streams. Following NVIDIA recommandations, we have first implemented a loop starting *NbS* asynchronous data transfers (lines 39–45):

transferring N/NbS data on each stream. Then we have implemented a second loop (lines 46–48), starting asynchronous executions of NbS grids of blocks of GPU threads (one per stream). Data transfers and kernel executions on the same stream are synchronized by CUDA and the GPU. So, each kernel execution will start after its data will be transferred into the GPU memory, and the GPU scheduler ensures to start some kernel executions as soon as the first data transfers are achieved. Then, next data transfers will be overlapped with GPU computations. After the kernel calls, on the different streams, we wait for the end of all GPU threads previously run, calling an explicit synchronization function on line 49. This synchronization is not mandatory, but it will make the implementation more robust and will facilitate the debugging steps: all GPU computations run by the OpenMP thread number 1 will be achieved before this thread will enter a new loop iteration, or before the computation loop will be ended.

If a partial result has to be transferred from GPU to CPU memory at the end of each loop iteration (for example the result of one *reduction* per iteration), this transfer is achieved synchronously on the default stream (no particular stream is specified) on lines 51–54. Availability of the result values is ensured by the synchronization implemented on line 49. However, if a partial result has to be transferred on the CPU on each stream, then NbS asynchronous data transfers could be started in parallel (one per stream), and should be implemented before the synchronization operation on line 49. The end of the computation loop includes a synchronization barrier of the two OpenMP threads, waiting they have finished to access the different data buffers in the current iteration. Then, each OpenMP thread exchanges its local buffer pointers, like in the previous algorithm. However, after the computation loop, we have added the destruction of the CUDA streams (lines 63–65).

Finally, CUDA streams have been used to extend Listing 7.2 with respect to its global scheme. Listing 7.3 still creates an OpenMP parallel region, with two CPU threads, one in charge of MPI communications, and the other managing data transfers and GPU computations. Unfortunately, using GPU streams require to be able to split a GPU computation in independent subparts, working on independent subsets of data. Listing 7.3 is not so generic than Listing 7.2.

7.2.4 Interleaved communications-transfers-computations overlapping

Many algorithms do not support to split data transfers and kernel calls, and can not exploit CUDA streams. For example, when each GPU thread requires to access some data spread in the global set of transferred data. Then, it is possible to overlap internode CPU communications and CPU/GPU data transfers and GPU computations, if the algorithm achieves *computation-communication iterations* and if we can interleave these iterations. At iteration k: CPUs exchange data D_k, each CPU/GPU couple transfers data D_k, and

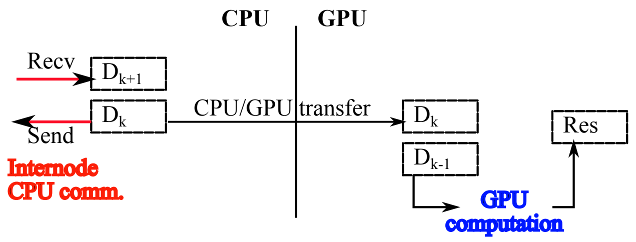

FIGURE 7.4: Complete overlap of internode CPU communications, CPU/GPU data transfers and GPU computations, interleaving computation-communication iterations

each GPU achieves computations on data D_{k-1} (see Figure 7.4). Compared to the previous strategies, this strategy requires twice as many CPU data buffers and twice as many GPU buffers.

Listing 7.4: Generic scheme explicitly overlapping MPI communications, CUDA CPU/GPU transfers and CUDA GPU computations, interleaving computation-communication iterations

```
// Input data and result variables and arrays (example with
// float datatype, 1D input arrays, and scalar results)
float *cpuInputTabAdrCurrent, *cpuInputTabAdrFuture;
float *gpuInputTabAdrCurrent, *gpuInputTabAdrFuture;
float *cpuResTabAdr, *gpuResAdr;

// CPU and GPU array allocations
cpuInputTabAdrCurrent = malloc(sizeof(float)*N);
cpuInputTabAdrFuture = malloc(sizeof(float)*N);
cudaMalloc(&gpuInputTabAdrCurrent,sizeof(float)*N);
cudaMalloc(&gpuInputTabAdrFuture,sizeof(float)*N);
cpuResTabAdr = malloc(sizeof(float)*NbIter);
cudaMalloc(&gpuResAdr,sizeof(float));

// Definition of the Grid of blocks of GPU threads
dim3 Dg, Db; Dg.x = ...
// Indexes of source and destination MPI processes
int dest, src; dest = ...

// Set the number of OpenMP threads (to create) to 2
omp_set_num_threads(3);
// Create threads and start the parallel OpenMP region
#pragma omp parallel
{
  // Buffer pointers (thread local variables)
  float *cpuCurrent = cpuInputTabAdrCurrent;
  float *cpuFuture  = cpuInputTabAdrFuture;
  float *gpuCurrent = gpuInputTabAdrCurrent;
  float *gpuFuture  = gpuInputTabAdrFuture;
  float *tmp;

  // Computation loop on: NbIter + 1 iteration
  for (int i = 0; i < NbIter + 1; i++) {
    // - Thread 0: achieves MPI communications
    if (omp_get_thread_num() == 0) {
      if (i < NbIter) {
        MPI_Sendrecv(cpuCurrent,                // MPI comms. (sync. op)
                     N, MPI_FLOAT, dest, 0,
                      cpuFuture,
40                    N, MPI_FLOAT, dest, 0, ...);
      }
    // - Thread 1: achieves the CPU/GPU data transfers
    } else if (omp_get_thread_num() == 1) {
      if (i < NbIter) {
45      cudaMemcpy(gpuFuture, cpuCurrent,     // Data transfer:
                   sizeof(float)*N,           // CPU --> GPU (sync. op
                       )
                   cudaMemcpyHostToDevice);
      }
    // - Thread 2: achieves the GPU computations and the result
        transfer
50  } else if (omp_get_thread_num() == 2) {
      if (i > 0) {
        gpuKernel_k1<<<Dg,Db>>>(gpuCurrent);  // GPU comp. (async. op)
        // IF there is (now) a result to transfer from GPU to CPU:
        cudaMemcpy(cpuResTabAdr + (i-1),      // Data transfer:
55                 gpuResAdr, sizeof(float),  // GPU --> CPU (sync. op
                       )
                   cudaMemcpyDeviceToHost);
      }
    }
    // - Wait for both threads have achieved their iteration tasks
60  #pragma omp barrier
    // - Each thread permute its local buffer pointers
    tmp = cpuCurrent; cpuCurrent = cpuFuture; cpuFuture = tmp;
    tmp = gpuCurrent; gpuCurrent = gpuFuture; gpuFuture = tmp;
  } // End of computation loop
65 } // End of OpenMP parallel region
...
```

Listing 7.4 introduces the generic code of a MPI+OpenMP+CUDA implementation, explicitly interleaving computation-communication iterations and overlapping MPI communications, CUDA CPU/GPU transfers and CUDA GPU computations. As in the previous algorithms, we declare two CPU input data arrays (current and future version) on line 3, but we also declare two GPU input data arrays on line 4. On lines 8–11, these four data arrays are allocated, using `malloc` and `cudaMalloc`. We do not need to allocate page-locked memory space. On lines 23–65 we create an OpenMP parallel region, configured to run three threads (see line 21). Lines 26–30 are declarations of thread local pointers on data arrays and variables (each thread will use its own pointers). On line 33, the three threads enter a computation loop of `NbIter + 1` iterations. We need to run one more iteration than with previous algorithms.

Lines 34–41 are the MPI communications, achieved by the thread number 0. They send the current CPU input data array to another CPU, and receive the future CPU input data array from another CPU, like in previous algorithms. But this thread achieves communications only during the *first* `NbIter` iterations. Lines 43–48 are the CPU to GPU input data transfers, achieved by thread number 1. These data transfers are run in parallel of MPI communications. They are run during the *first* `NbIter` iterations, and transfer current CPU input data array into the future GPU data array. Lines 50–57 correspond to the code run by thread number 3. They start GPU computations, to process the current GPU input data array, and if necessary transfer a GPU result at an index of the CPU result array. These GPU computations and result transfers are run during the *last* `NbIter` iterations: the GPU computations have to wait the first data transfer is ended before to start to process any data, and can not run during the first iteration. So, the activity of the third thread is shifted of one iteration compared to the activities of other threads. Moreover, the address of the current GPU input data array has to be passed as a parameter of the kernel call on line 52, in order the GPU threads access the right data array. Like in previous algorithms the GPU result is copied at one index of the CPU result array, in lines 53–56, but due to the shift of the third thread activity this index is now `(i - 1)`.

Line 60 is a synchronization barrier of the three OpenMP threads, followed by a pointer permutation of local pointers on current and future data arrays, on line 62 and 63. Each thread waits for the completion of other threads to use the data arrays, and then permutes its data array pointers before to enter a new loop iteration.

This complete overlap of MPI communications and CPU/GPU data transfers and GPU computations, is not too complex to implement, and can be a solution when GPU computations are not adapted to use CUDA streams: when GPU computations can not be split in subparts working on independent subsets of input data. However, it requires to run one more iterations (a total of `NbIter + 1` iterations). Then, if the number of iterations is very small, it could be more interesting not to attempt to overlap CPU/GPU data transfers and GPU computations, and to implement Listing 7.2.

7.2.5 Experimental validation

Experimentation testbed

Two clusters located at SUPELEC in Metz (France) have been used for the entire set of experiments presented in this chapter:

- The first consists of 17 nodes with an Intel Nehalem quad-core processor at 2.67Ghz, 6 Gb RAM and an NVIDIA GeForce GTX480 GPU, each.
- The second consists of 16 nodes with an Intel core2 dual-core processor at 2.67Ghz, 4 Gb RAM and an NVIDIA GeForce GTX580 GPU, each

Both clusters have a Gigabit Ethernet interconnection network that is connected through a DELL Power Object 5324 switch. The two switches are linked twice, insuring the interconnection of the two clusters. The software environment consists of a Linux Fedora 64bit OS (kernel v. 2.6.35), GNU C and C++ compilers (v. 4.5.1) and the CUDA library (v. 4.2).

Validation of the synchronous approach

We have experimented our approach of synchronous parallel algorithms with a classic block cyclic algorithm for dense matrix multiplication. This

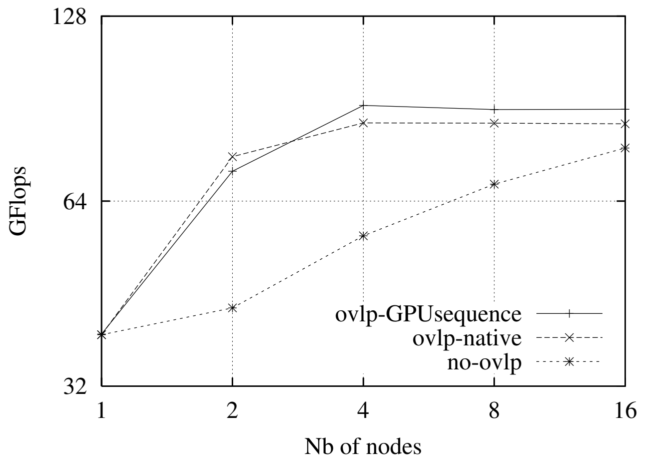

FIGURE 7.5: Experimental performances of different synchronous algorithms computing a dense matrix product

problem requires to split two input matrices (A and B) on a ring of computing nodes, and to establish a circulation of the slices of A matrix on the ring (B matrix partition does not evolve during all the run). Compared to our generic algorithms, there is no partial result to transfer from GPU to CPU at the end of each computing iteration. The part of the result matrix computed on each GPU is transferred on the CPU at the end of the computation loop.

We have first implemented a synchronous version without any overlap of MPI communications, CPU/GPU data transfers, and GPU computations. We have added some synchronizations in the native overlapping version in order to avoid any overlap. We have measured the performance achieved on our cluster with NVIDIA GTX480 GPUs and matrices sizes of 4096×4096, and we have obtained the curves in Figure 7.5 labeled *no-ovlp*. We observe that performance increases when the number of processor increases. Of course, there is a significant increase in cost when comparing a single node (without any MPI communication) with two nodes (starting to use MPI communications). But beyond two nodes we get a classical performance curve.

Then, we implemented and experimented Listing 7.1, see *ovlp-native* in Figure 7.5. The native overlap of MPI communications with asynchronous run of CUDA kernels appears efficient. When the number of nodes increases the ratio of the MPI communications increases a lot (because the computation times decrease a lot). So, there is not a lot of GPU computation time that remains to be overlapped, and both *no-ovlp* and *ovlp-native* tend to the same limit. Already, the native overlap performed in Listing 7.1 achieves a high level of performance very quickly, using only four nodes. Beyond four nodes, a faster interconnection network would be required for a performance increase.

Finally, we implemented Listing 7.2, overlapping MPI communications with a GPU sequence including both CPU/GPU data transfers and GPU computations, see *ovlp-GPUsequence* in Figure 7.5. From four up to sixteen nodes it achieves better performances than *ovlp-native*: we better overlap MPI communications. However, this parallelization mechanism has more overhead: OpenMP threads have to be created and synchronized. Only for two nodes it is less efficient than the native overlapping algorithm. Beyond two nodes, the CPU multithreading overhead seems compensated. Listing 7.2 requires more time for the implemention and can be more complex to maintain, but such extra development cost is justified if we are looking for better performance.

7.3 General scheme of asynchronous parallel code with computation/communication overlapping

In the previous part, we have seen how to efficiently implement overlap of computations (CPU and GPU) with communications (GPU transfers and inter-node communications). However, we have previously shown that for some parallel iterative algorithms, it is sometimes even more efficient to use an asynchronous scheme of iterations [3, 4, 11]. In that case, the nodes do not wait for each others but they perform their iterations using the last external data they have received from the other nodes, even if this data was produced *before* the previous iteration on the other nodes.

Formally, if we denote by $f = (f_1, ..., f_n)$ the function representing the iterative process and by $x^t = (x_1^t, ..., x_n^t)$ the values of the n elements of the system at iteration t, we pass from a synchronous iterative scheme of the form:

Algorithm 4: Synchronous iterative scheme

```
1 $x^0 = (x_1^0, ..., x_n^0)$;
2 for $t = 0, 1, ...$ do
3     for $i = 1, ..., n$ do
4         $x_i^{t+1} = f_i(x_1^t, ..., x_i^t, ..., x_n^t)$;
5     end
6 end
```

to an asynchronous iterative scheme of the form:

Algorithm 5: Asynchronous iterative scheme

```
1 $x^0 = (x_1^0, ..., x_n^0)$;
2 for $t = 0, 1, ...$ do
3     for $i = 1, ..., n$ do
4         $x_i^{t+1} = \begin{cases} x_i^t & \text{if } i \text{ is } \textit{not} \text{ updated at iteration } i \\ f_i(x_1^{s_1^i(t)}, ..., x_n^{s_n^i(t)}) & \text{if } i \text{ is updated at iteration } i \end{cases}$
5     end
6 end
```

where $s_j^i(t)$ is the iteration number of the production of the value x_j of element j that is used on element i at iteration t (see for example [9, 12] for further details). Such schemes are called AIAC for *Asynchronous Iterations and Asynchronous Communications*. They combine two aspects that are respectively different computation speeds of the computing elements and communication delays between them.

The key feature of such algorithmic schemes is that they may be faster than their synchronous counterparts due to the implied total overlap of computations with communications: in fact, this scheme suppresses all the idle times induced by nodes synchronizations between each iteration.

However, the efficiency of such a scheme is directly linked to the frequency at which new data arrives on each node. Typically, if a node receives newer data only every four or five local iterations, it is strongly probable that the evolution of its local iterative process will be slower than if it receives data at every iteration. The key point here is that this frequency does not only depend on the hardware configuration of the parallel system but it also depends on the software that is used to implement the algorithmic scheme.

The impact of the programming environments used to implement asynchronous algorithms has already been investigated in [5]. Although the features required to efficiently implement asynchronous schemes have not changed, the available programming environments and computing hardware have evolved, in particular now GPUs are available. So, there is a need to reconsider the implementation schemes of AIAC according to the new de facto standards for parallel programming (communications and threads) as well as the integration of the GPUs. One of the main objective here is to obtain a maximal overlap between the activities of the three types of devices that are the CPU, the GPU and the network. Moreover, another objective is to present what we think is the best compromise between the simplicity of the implementation and its maintainability on one side and its performance on the other side. This is especially important for industries where implementation and maintenance costs are strong constraints.

For the sake of clarity, we present the different algorithmic schemes in a progressive order of complexity, from the basic asynchronous scheme to the complete scheme with full overlap. Between these two extremes, we propose a synchronization mechanism on top of our asynchronous scheme that can be used either statically or dynamically during the application execution.

Although there exist several programming environments for inter-node communications, multi-threading and GPU programming, a few of them have become *de facto standards*, either due to their good stability, their ease of use and/or their wide adoption by the scientific community. Therefore, as in the previous section all the schemes presented in the following use MPI [1], OpenMP [2] and CUDA [18]. However, there is no loss of generality as those schemes may easily be implemented with other libraries.

Finally, in order to stay as clear as possible, only the parts of code and variables related to the control of parallelism (communications, threads,...) are presented in our schemes. The inner organization of data is not detailed as it depends on the application. We only consider that we have two data arrays (previous version and current version) and communication buffers. However, in most of the cases, those buffers can correspond to the data arrays themselves to avoid data copies.

7.3.1 A basic asynchronous scheme

The first step toward our complete scheme is to implement a basic asynchronous scheme that includes an actual overlap of the communications with the computations. In order to ensure that the communications are actually performed in parallel of the computations, it is necessary to use different threads. It is important to remember that asynchronous communications provided in communication libraries like MPI are not systematically performed in parallel of the computations [15, 21]. So, the logical and classical way to implement such an overlap is to use three threads: one for computing, one for sending and one for receiving. Moreover, since the communication is performed by threads, blocking synchronous communications can be used without deteriorating the overall performance.

In this basic version, the termination of the global process is performed individually on each node according to their own termination. This can be guided by either a number of iterations or a local convergence detection. The important step at the end of the process is to perform the receptions of all pending communications in order to ensure the termination of the two communication threads.

So, the global organization of this scheme is set up in Listing 7.5.

Listing 7.5: Initialization of the basic asynchronous scheme

```
// Variables declaration and initialization
omp_lock_t lockSend; // Controls the sendings from the computing
    thread
omp_lock_t lockRec;  // Ensures the initial reception of external data
char Finished = 0;   // Boolean indicating the end of the process
char SendsInProgress = 0; // Boolean indicating if previous data
    sendings are still in progress
double Threshold;    // Threshold of the residual for convergence
    detection
```

```
   // Parameters reading
   ...
10
   // MPI initialization
   MPI_Init_thread(argc, argv, MPI_THREAD_MULTIPLE, &provided);
   MPI_Comm_size(MPI_COMM_WORLD, &nbP);
   MPI_Comm_rank(MPI_COMM_WORLD, &numP);
15
   // Data initialization and distribution among the nodes
   ...

   // OpenMP initialization (mainly declarations and setting up of locks)
20 omp_set_num_threads(3);
   omp_init_lock(&lockSend);
   omp_set_lock(&lockSend); // Initially locked, unlocked to start
       sendings
   omp_init_lock(&lockRec);
   omp_set_lock(&lockRec);  // Initially locked, unlocked when initial
       data are received
25
   #pragma omp parallel
   {
     switch(omp_get_thread_num()){
       case COMPUTATION :
30     computations(... @\emph{relevant parameters}@ ...);
       break;

       case SENDINGS :
       sendings();
35     break;

       case RECEPTIONS :
       receptions();
       break;
40   }
   }

   // Cleaning of OpenMP locks
   omp_test_lock(&lockSend);
45 omp_unset_lock(&lockSend);
   omp_destroy_lock(&lockSend);
   omp_test_lock(&lockRec);
   omp_unset_lock(&lockRec);
   omp_destroy_lock(&lockRec);
50
   // MPI termination
   MPI_Finalize();
```

In this scheme, the `lockRec` mutex is not mandatory. It is only used to ensure that data dependencies are actually exchanged at the first iteration of the process. Data initialization and distribution (lines 16-17) are not detailed here because they are directly related to the application. The important point is that, in most cases, they should be done before the iterative process. The computing function is given in Listing 7.6.

Listing 7.6: Computing function in the basic asynchronous scheme

```
// Variables declaration and initialization
int iter = 1;        // Number of the current iteration
double difference;   // Variation of one element between two iterations
double residual;     // Residual of the current iteration

// Computation loop
```

```
   while(!Finished){
     // Sendings of data dependencies if there is no previous sending in
         progress
     if(!SendsInProgress){
10     // Potential copy of data to be sent in additional buffers
       ...
       // Change of sending state
       SendsInProgress = 1;
       omp_unset_lock(&lockSend);
15   }

     // Blocking receptions at the first iteration
     if(iter == 1){
       omp_set_lock(&lockRec);
20   }

     // Initialization of the residual
     residual = 0.0;
     // Swapping of data arrays (current and previous)
25   tmp = current;      // Pointers swapping to avoid
     current = previous; // actual data copies between
     previous = tmp;     // the two data versions
     // Computation of current iteration over local data
     for(ind=0; ind<localSize; ++ind){
30     // Updating of current array using previous array
       ...
       // Updating of the residual
       // (max difference between two successive iterations)
       difference = fabs(current[ind] - previous[ind]);
35     if(difference > residual){
         residual = difference;
       }
     }

40   // Checking of the end of the process (residual under threshold)
     // Other conditions can be added to the termination detection
     if(residual <= Threshold){
       Finished = 1;
       omp_unset_lock(&lockSend); // Activation of end messages sendings
45     MPI_Ssend(&Finished, 1, MPI_CHAR, numP, tagEnd, MPI_COMM_WORLD);
     }

     // Updating of the iteration number
     iter++;
50 }
```

As mentioned above, it can be seen in line 18 of Listing 7.6 that the `lockRec` mutex is used only at the first iteration to wait for the initial data dependencies before the computations. The residual, initialized in line 23 and computed in lines 34-37, is defined by the maximal difference between the elements from two consecutive iterations. It is classically used to detect the local convergence of the process on each node. In the more complete schemes presented in the sequel, a global termination detection that takes the states of all the nodes into account will be exhibited.

Finally, the local convergence is tested and updated when necessary. In line 44, the `lockSend` mutex is unlocked to allow the sending function to send final messages to the dependency nodes. Those messages are required to keep the reception function alive until all the final messages have been received. Otherwise, a node could stop its reception function whereas other nodes are still trying to communicate with it. Moreover, a local sending of a final message to the node itself is required (line 45) to ensure that the reception function will not stay blocked in a message probing (see Listing 7.8, line 11). This may happen if the node receives the final messages from its dependencies *before* being itself in local convergence.

All the messages but this final local one are performed in the sending function described in Listing 7.7.

The main loop is only conditioned by the end of the computing process (line 4). At each iteration, the thread waits for the permission from the computing thread (according to the `lockSend` mutex). Then, data are sent with blocking synchronous communications. The `SendsInProgress` boolean allows the computing thread to skip data sendings as long as a previous sending is in progress. This skip is possible due to the nature of asynchronous algorithms that allows such *message loss* or *message miss*. After the main loop, the final messages are sent to the dependencies of the node.

Listing 7.7: Sending function in the basic asynchronous scheme

```
   // Variables declaration and initialization
   ...

   while(!Finished){
5    omp_set_lock(&lockSend); // Waiting for signal from the comp. thread
     if(!Finished){
       // Blocking synchronous sends to all dependencies
       for(i=0; i<nbDeps; ++i){
         MPI_Ssend(&dataToSend[deps[i]], nb_data, type_of_data, deps[i],
              tagCom, MPI_COMM_WORLD);
10     }
       SendsInProgress = 0; // Indicates that the sendings are done
     }
   }
   // At the end of the process, sendings of final messages
15 for(i=0; i<nbDeps; ++i){
     MPI_Ssend(&Finished, 1, MPI_CHAR, deps[i], tagEnd, MPI_COMM_WORLD);
   }
```

The last function, detailed in Listing 7.8, does all the messages receptions.

Listing 7.8: Reception function in the basic asynchronous scheme

```
   // Variables declaration and initialization
   char countReceipts = 0; // Boolean indicating whether receptions are
       counted or not
   int nbEndMsg = 0;       // Number of end messages received
   int arrived = 0;        // Boolean indicating if a message is arrived
5  int srcNd;              // Source node of the message
   int size;               // Message size

   // Main loop of receptions
   while(!Finished){
10   // Waiting for an incoming message
     MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
     if(!Finished){
       // Management of data messages
       switch(status.MPI_TAG){
15       case tagCom: // Management of data messages
          srcNd = status.MPI_SOURCE; // Get the source node of the
              message
          // Actual data reception in the corresponding buffer
          MPI_Recv(dataBufferOf(srcNd), nbDataOf(srcNd), dataTypeOf(srcNd
              ), srcNd, tagCom, MPI_COMM_WORLD, &status);
          // Unlocking of the computing thread when data are received
              from all dependencies
20        if(countReceipts == 1 && ... @\emph{receptions from ALL
              dependencies}@ ...){
            omp_unset_lock(&lockRec);
            countReceipts = 0; // No more counting after first iteration
          }
          break;
25      case tagEnd: // Management of end messages
          // Actual end message reception in dummy buffer
          MPI_Recv(dummyBuffer, 1, MPI_CHAR, status.MPI_SOURCE, tagEnd,
              MPI_COMM_WORLD, &status);
          nbEndMsg++;
      }
30  }
}

// Reception of pending messages and counting of end messages
do{ // Loop over the remaining incoming/waited messages
35  MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_CHAR, &size);
    // Actual reception in dummy buffer
    MPI_Recv(dummyBuffer, size, MPI_CHAR, status.MPI_SOURCE, status.
        MPI_TAG, MPI_COMM_WORLD, &status);
    if(status.MPI_TAG == tagEnd){ // Counting of end messages
40    nbEndMsg++;
    }
    MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &arrived, &
        status);
}while(arrived == 1 || nbEndMsg < nbDeps + 1);
```

As in the sending function, the main loop of receptions is done while the iterative process is not `Finished`. In line 11, the thread waits until a message arrives on the node. Then, it performs the actual reception and the corresponding subsequent actions (potential data copies for data messages and counting for end messages). Lines 20-23 check that all data dependencies have been received before unlocking the `lockRec` mutex. As mentioned before, they are not mandatory and are included only to ensure that all data dependencies are received at the first iteration. Lines 25-28 are required to manage end messages that arrive on the node *before* it reaches its own termination process. As the nodes are *not* synchronized, this may happen. Finally, lines 34-43 perform the receptions of all pending communications, including the remaining end messages (at least the one from the node itself).

So, with those algorithms, we obtain a quite simple and efficient asynchronous iterative scheme. It is interesting to notice that GPU computing can be easily included in the computing thread. This will be fully addressed in paragraph 7.3.3. However, before presenting the complete asynchronous scheme with GPU computing, we have to detail how our initial scheme can be made synchronous.

7.3.2 Synchronization of the asynchronous scheme

The presence of synchronization in the previous scheme may seem contradictory to our goal, and obviously, it is neither the simplest way to obtain a synchronous scheme nor the most efficient (as presented in Section 7.2). However, it is necessary for our global convergence detection strategy. Recall that the global convergence is the extension of the local convergence concept to all the nodes. This implies that all the nodes have to be in local convergence at the same time to achieve global convergence. Typically, if we use the residual and a threshold to stop the iterative process, all the nodes have to continue their local iterative process until *all* of them obtain a residual under the threshold.

In our context, the interest of being able to dynamically change the operating mode (sync/async) during the process execution, is that this strongly simplifies the global convergence detection. In fact, our past experience in the design and implementation of global convergence detection in asynchronous algorithms [5–7], have led us to the conclusion that although a decentralized detection scheme is possible and may be more efficient in some situations, its much higher complexity is an obstacle to actual use in practice, especially in industrial contexts where implementation/maintenance costs are strong constraints. Moreover, although the decentralized scheme does not slow down the computations, it requires more iterations than a synchronous version and thus may induce longer detection times in some cases. So, the solution we present below is a good compromise between simplicity and efficiency. It consists in dynamically changing the operating mode between asynchronous and synchronous during the execution of the process in order to check the global convergence. This is why we need to synchronize our asynchronous scheme.

In each algorithm of the initial scheme, we only exhibit the additional code required to change the operating mode.

Listing 7.9: Initialization of the synchronized scheme

```
// Variables declarations and initialization
...
omp_lock_t lockStates; // Controls the synchronous exchange of local
    states
omp_lock_t lockIter;   // Controls the synchronization at the end of
    each iteration
char localCV = 0;      // Boolean indicating whether the local
    stabilization is reached or not
int nbOtherCVs = 0;    // Number of other nodes being in local
    stabilization

// Parameters reading
...
// MPI initialization
...
// Data initialization and distribution among the nodes
...
// OpenMP initialization (mainly declarations and setting up of locks)
...
omp_init_lock(&lockStates);
omp_set_lock(&lockStates); // Initially locked, unlocked when all
    state messages are received
```

```
omp_init_lock(&lockIter);
omp_set_lock(&lockIter);   // Initially locked, unlocked when all "end
     of iteration" messages are received

// Threads launching
#pragma omp parallel
{
  switch(omp_get_thread_num()){
    ...
  }
}

// Cleaning of OpenMP locks
...
omp_test_lock(&lockStates);
omp_unset_lock(&lockStates);
omp_destroy_lock(&lockStates);
omp_test_lock(&lockIter);
omp_unset_lock(&lockIter);
omp_destroy_lock(&lockIter);

// MPI termination
MPI_Finalize();
```

As can be seen in Listing 7.9, the synchronization implies two additional mutex. The `lockStates` mutex is used to wait for the receptions of all state messages coming from the other nodes. As shown in Listing 7.10, those messages contain only a boolean indicating for each node if it is in local convergence. So, once all the states are received on a node, it is possible to determine if all the nodes are in local convergence, and thus to detect the global convergence. The `lockIter` mutex is used to synchronize all the nodes at the end of each iteration. There are also two new variables that respectively represent the local state of the node (`localCV`) according to the iterative process (convergence) and the number of other nodes that are in local convergence (`nbOtherCVs`).

The computation thread is where most of the modifications take place, as shown in Listing 7.10.

Listing 7.10: Computing function in the synchronized scheme

```
// Variables declarations and initialization
...

// Computation loop
while(!Finished){
  // Sendings of data dependencies at @\emph{each}@ iteration
  // Potential copy of data to be sent in additional buffers
  ...
  omp_unset_lock(&lockSend);

  // Blocking receptions at @\emph{each}@ iteration
  omp_set_lock(&lockRec);

  // Local computation
  // (init of residual, arrays swapping and iteration computation)
  ...

  // Checking of the stabilization of the local process
  // Other conditions than the residual can be added
  if(residual <= Threshold){
      localCV = 1;
    }else{
      localCV = 0;
    }
25
    // Global exchange of local states of the nodes
    for(ind=0; ind<nbP; ++ind){
      if(ind != numP){
        MPI_Ssend(&localCV, 1, MPI_CHAR, ind, tagState, MPI_COMM_WORLD);
30    }
    }

    // Waiting for the state messages receptions from the other nodes
    omp_set_lock(&lockStates);
35
    // Determination of global convergence (if all nodes are in local CV
       )
    if(localCV + nbOtherCVs == nbP){
      // Entering global CV state
      Finished = 1;
40    // Unlocking of sending thread to start sendings of end messages
      omp_unset_lock(&lockSend);
      MPI_Ssend(&Finished, 1, MPI_CHAR, numP, tagEnd, MPI_COMM_WORLD);
    }else{
      // Resetting of information about the states of the other nodes
45    ...
      // Global barrier at the end of each iteration during the process
      for(ind=0; ind<nbP; ++ind){
        if(ind != numP){
          MPI_Ssend(&Finished, 1, MPI_CHAR, ind, tagIter, MPI_COMM_WORLD
              );
50      }
      }
      omp_set_lock(&lockIter);
    }

55
    // Updating of the iteration number
    iter++;
}
```

Most of the added code is related to the waiting for specific communications. Between lines 6 and 7, the use of the flag `SendsInProgress` is no longer needed since the sends are performed at each iteration. In line 12, the thread waits for the data receptions from its dependencies. In lines 26-34, the local states are determined and exchanged among all nodes. A new message tag (`tagState`) is required for identifying those messages. In line 37, the global termination state is determined. When it is reached, lines 38-42 change the `Finished` boolean to stop the iterative process, and send the end messages. Otherwise each node resets its local state information about the other nodes and a global barrier is done between all the nodes at the end of each iteration with another new tag (`tagIter`). That barrier is needed to ensure that data messages from successive iterations are actually received during the *same* iteration on the destination nodes. Nevertheless, it is not useful at the termination of the global process as it is replaced by the global exchange of end messages.

There is no big modification induced by the synchronization in the sending function. The only change could be the suppression of line 11 that is not useful in this case. Apart from that, the function stays the same as in Listing 7.7.

In the reception function, given in Listing 7.11, there are mainly two insertions (in lines 19-30 and 31-40), corresponding to the additional types of messages to receive. There is also the insertion of three variables that are used for the receptions of the new message types. In lines 24-29 and 34-39 are located messages counting and mutex unlocking mechanisms that are used to block the computing thread at the corresponding steps of its execution. They are similar to the mechanism used for managing the end messages at the end of the entire process. Line 23 directly updates the number of other nodes that are in local convergence by adding the received state of the source node. This is possible due to the encoding that is used to represent the local convergence (1) and the non-convergence (0).

Listing 7.11: Reception function in the synchronized scheme

```
   // Variables declarations and initialization
   ...
   int nbStateMsg = 0; // Number of local state messages received
   int nbIterMsg = 0;  // Number of "end of iteration" messages received
 5 char recvdState;    // Received state from another node (0 or 1)

   // Main loop of receptions
   while(!Finished){
     // Waiting for an incoming message
10   MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
     if(!Finished){
       switch(status.MPI_TAG){ // Actions related to message type
         case tagCom: // Management of data messages
          ...
15        break;
         case tagEnd: // Management of termination messages
          ...
          break;
         case tagState: // Management of local state messages
20        // Actual reception of the message
          MPI_Recv(&recvdState, 1, MPI_CHAR, status.MPI_SOURCE, tagState,
               MPI_COMM_WORLD, &status);
          // Updates of numbers of stabilized nodes and received state
              msgs
          nbOtherCVs += recvdState;
          nbStateMsg++;
25        // Unlocking of the computing thread when states of all other
              nodes are received
          if(nbStateMsg == nbP-1){
            nbStateMsg = 0;
            omp_unset_lock(&lockStates);
          }
30        break;
         case tagIter: // Management of "end of iteration" messages
          // Actual reception of the message in dummy buffer
          MPI_Recv(dummyBuffer, 1, MPI_CHAR, status.MPI_SOURCE, tagIter,
              MPI_COMM_WORLD, &status);
          nbIterMsg++; // Update of the nb of iteration messages
35        // Unlocking of the computing thread when iteration messages
              are received from all other nodes
          if(nbIterMsg == nbP - 1){
            nbIterMsg = 0;
            omp_unset_lock(&lockIter);
          }
40        break;
```

```
    }
  }
}

// Reception of pending messages and counting of end messages
do{ // Loop over the remaining incoming/waited messages
  ...
}while(arrived == 1 || nbEndMsg < nbDeps + 1);
```

Now that we can synchronize our asynchronous scheme, the final step is to dynamically alternate the two operating modes in order to regularly check the global convergence of the iterative process. This is detailed in the following paragraph together with the inclusion of GPU computing in the final asynchronous scheme.

7.3.3 Asynchronous scheme using MPI, OpenMP and CUDA

As mentioned above, the strategy proposed to obtain a good compromise between simplicity and efficiency in the asynchronous scheme is to dynamically change the operating mode of the process. A good way to obtain a maximal simplification of the final scheme while preserving good performance is to perform local and global convergence detections only in synchronous mode. Moreover, as two successive iterations are sufficient in synchronous mode to detect local and global convergences, the key is to alternate some asynchronous iterations with two synchronous iterations until convergence.

The last problem is to decide *when* to switch from the asynchronous to the synchronous mode. Here again, for the sake of simplicity, any asynchronous mechanism for *detecting* such instant is avoided and we prefer to use a mechanism that is local to each node. Obviously, that local system must rely neither on the number of local iterations done nor on the local convergence. The former would slow down the fastest nodes according to the slowest ones. The latter would provoke too much synchronization because the residuals on all nodes commonly do not evolve in the same way and, in most cases, there is a convergence wave phenomenon throughout the elements. So, a good solution is to insert a local timer mechanism on each node with a given initial duration. Then, that duration may be modified during the execution according to the successive results of the synchronous sections.

Another problem induced by entering synchronous mode from the asynchronous one is the possibility to receive some data messages from previous asynchronous iterations during synchronous iterations. This could lead to deadlocks. In order to avoid this, a wait of the end of previous send is added to the transition between the two modes. This is implemented by replacing the variable `SendsInProgress` by a mutex `lockSendsDone` which is unlocked once all the messages have been sent in the sending function. Moreover, it is also necessary to stamp data messages (by the function `stampData`) with a Boolean indicating whether they have been sent during a synchronous or asynchronous iteration. Then, the `lockRec` mutex is unlocked only after to the complete reception of data messages from synchronous iterations. The message ordering of point-to-point communications in MPI together with the barrier at the end of each iteration ensure two important properties of this mechanism. First, data messages from previous asynchronous iterations will be received but not taken into account during synchronous sections. Then, a data message from a synchronous iteration cannot be received in another synchronous iteration. In the asynchronous sections, no additional mechanism is needed as there are no such constraints concerning the data receptions.

Finally, the required modifications of the previous scheme are mainly related to the computing thread. Small additions or modifications are also required in the main process and the other threads.

In the main process, two new variables are added to store respectively the main operating mode of the iterative process (`mainMode`) and the duration of asynchronous sections (`asyncDuration`). Those variables are initialized by the programmer. The `lockSendsDone` mutex is also declared, initialized (locked) and destroyed with the other mutex in this process.

In the computing function, shown in Listing 7.12, the modifications consist of the insertion of the timer mechanism and of the conditions to differentiate the actions to be done in each mode. Some additional variables are also required to store the current operating mode in action during the execution (`curMode`), the starting time of the current asynchronous section (`asyncStart`) and the number of successive synchronous iterations done (`nbSyncIter`).

Listing 7.12: Computing function in the final asynchronous scheme

```
// Variables declarations and initialization
...
OpMode curMode = SYNC; // Current operating mode (always begin in sync)
double asyncStart;      // Starting time of the current async section
int nbSyncIter = 0;     // Number of sync iterations done in async mode

// Computation loop
while(!Finished){
  // Determination of the dynamic operating mode
  if(curMode == ASYNC){
    // Entering synchronous mode when asyncDuration is reached
@%    // (additional conditions can be specified if needed)
@    if(MPI_Wtime() - asyncStart >= asyncDuration){
      // Waiting for the end of previous sends before starting sync mode
      omp_set_lock(&lockSendsDone);
      curMode = SYNC;               // Entering synchronous mode
      stampData(dataToSend, SYNC); // Mark data to send with sync flag
      nbSyncIter = 0;
    }
  }else{
    // In main async mode, going back to async mode when the max number of sync iterations are done
    if(mainMode == ASYNC){
      nbSyncIter++; // Update of the number of sync iterations done
      if(nbSyncIter == 2){
        curMode = ASYNC;                // Going back to async mode
        stampData(dataToSend, ASYNC); // Mark data to send
        asyncStart = MPI_Wtime();      // Get the async starting time
```

```
            }
        }
30  }

    // Sendings of data dependencies
    if(curMode == SYNC || !SendsInProgress){
      ...
35  }

    // Blocking data receptions in sync mode
    if(curMode == SYNC){
      omp_set_lock(&lockRec);
40  }

    // Local computation
    // (init of residual, arrays swapping and iteration computation)
    ...
45
    // Checking of convergences (local & global) only in sync mode
    if(curMode == SYNC){
      // Local convergence checking (residual under threshold)
      ...
50    // Blocking global exchange of local states of the nodes
      ...
      // Determination of global convergence (all nodes in local CV)
      //    Stop of the iterative process and sending of end messages
      // or Re-initialization of state information and iteration barrier
55    ...
      }
    }

    // Updating of the iteration number
60  iter++;
  }
```

In the sending function, the only modification is the replacement in line 11 of the assignment of variable `SendsInProgress` by the unlocking of `lockSendsDone`. Finally, in the reception function, the only modification is the insertion before line 19 of Listing 7.8 of the extraction of the stamp from the message and its counting among the receipts only if the stamp is `SYNC`.

The final step to get our complete scheme using GPU is to insert the GPU management in the computing thread. The first possibility, detailed in Listing 7.13, is to simply replace the CPU kernel (lines 41-43 in Listing 7.12) by a blocking GPU kernel call. This includes data transfers from the node RAM to the GPU RAM, the launching of the GPU kernel, the waiting for kernel completion and the results transfers from GPU RAM to node RAM.

Listing 7.13: Computing function in the final asynchronous scheme

```
// Variables declarations and initialization
...
dim3 Dg, Db; // CUDA kernel grids

// Computation loop
while(!Finished){
  // Determination of the dynamic operating mode, sendings of data
      dependencies and blocking data receptions in sync mode
  ...
  // Local GPU computation
  // Data transfers from node RAM to GPU
  CHECK_CUDA_SUCCESS(cudaMemcpyToSymbol(dataOnGPU, dataInRAM,
      inputsSize, 0, cudaMemcpyHostToDevice), "Data transfer");
```

```
     ... // There may be several data transfers: typically A and b in
         linear problems
     // GPU grid definition
     Db.x = BLOCK_SIZE_X; // BLOCK_SIZE# are kernel design dependent
15   Db.y = BLOCK_SIZE_Y;
     Db.z = BLOCK_SIZE_Z;
     Dg.x = localSize/BLOCK_SIZE_X + (localSize%BLOCK_SIZE_X ? 1 : 0);
     Dg.y = localSize/BLOCK_SIZE_Y + (localSize%BLOCK_SIZE_Y ? 1 : 0);
     Dg.z = localSize/BLOCK_SIZE_Z + (localSize%BLOCK_SIZE_Z ? 1 : 0);
20   // Use of shared memory (when possible)
     cudaFuncSetCacheConfig(gpuKernelName, cudaFuncCachePreferShared);
     // Kernel call
     gpuKernelName<<<Dg,Db>>>(... @\emph{kernel parameters}@ ...);
     // Waiting for kernel completion
25   cudaDeviceSynchronize();
     // Results transfer from GPU to node RAM
     CHECK_CUDA_SUCCESS(cudaMemcpyFromSymbol(resultsInRam, resultsOnGPU,
         resultsSize, 0, cudaMemcpyDeviceToHost), "Results␣transfer");
     // Potential post-treatment of results on the CPU
     ...
30
     // Convergences checking
     ...
}
```

This scheme provides asynchronism through a cluster of GPUs as well as a complete overlap of communications with GPU computations (similarly to Section 7.2). However, the autonomy of GPU devices according to their host can be further exploited in order to perform some computations on the CPU while the GPU kernel is running. The nature of computations that can be done by the CPU may vary depending on the application. For example, when processing data streams (pipelines), pre-processing of next data item and/or post-processing of previous result can be done on the CPU while the GPU is processing the current data item. In other cases, the CPU can perform *auxiliary* computations that are not absolutely required to obtain the result but that may accelerate the entire iterative process. Another possibility would be to distribute the main computations between the GPU and CPU. However, this usually leads to poor performance increases. This is mainly due to data dependencies that often require additional transfers between CPU and GPU.

So, if we consider that the application enables such overlap of computations, its implementation is straightforward as it consists in inserting the additional CPU computations between lines 23 and 24 in Listing 7.13. Nevertheless, such scheme is fully efficient only if the computation times on both sides are similar.

In some cases, especially with auxiliary computations, another interesting solution is to add a fourth CPU thread to perform them. This suppresses the duration constraint over those optional computations as they are performed in parallel of the main iterative process, without blocking it. Moreover, this scheme stays coherent with current architectures as most nodes include four CPU cores. The algorithmic scheme of such context of complete overlap of CPU/GPU computations and communications is described in Listings 7.14, 7.15 and 7.16, where we suppose that auxiliary computations use intermediate results of the main computation process from any previous iteration. This may be different according to the application.

Listing 7.14: Initialization of the main process of complete overlap with asynchronism

```
// Variables declarations and initialization
...
omp_lock_t lockAux;   // Informs main thread about new aux results
omp_lock_t lockRes;   // Informs aux thread about new results
omp_lock_t lockWrite; // Controls exclusion of results access
... auxRes ... ;      // Results of auxiliary computations

// Parameters reading, MPI initialization, data initialization and
    distribution
...
// OpenMP initialization
...
omp_init_lock(&lockAux);
omp_set_lock(&lockAux);  // Unlocked when new aux results are
    available
omp_init_lock(&lockRes);
omp_set_lock(&lockRes);  // Unlocked when new results are available
omp_init_lock(&lockWrite);
omp_unset_lock(&lockWrite); // Controls access to results from threads

#pragma omp parallel
{
  switch(omp_get_thread_num()){
    case COMPUTATION :
    computations(... @\emph{relevant parameters}@ ...);
    break;

    case AUX_COMPS :
    auxComps(... @\emph{relevant parameters}@ ...);
    break;

    case SENDINGS :
    sendings();
    break;

    case RECEPTIONS :
    receptions();
    break;
  }
}

// Cleaning of OpenMP locks
...
omp_test_lock(&lockAux);
omp_unset_lock(&lockAux);
omp_destroy_lock(&lockAux);
omp_test_lock(&lockRes);
omp_unset_lock(&lockRes);
omp_destroy_lock(&lockRes);
omp_test_lock(&lockWrite);
omp_unset_lock(&lockWrite);
omp_destroy_lock(&lockWrite);

// MPI termination
MPI_Finalize();
```

Listing 7.15: Computing function in the final asynchronous scheme with CPU/GPU overlap

```
// Variables declarations and initialization
...
dim3 Dg, Db; // CUDA kernel grids

// Computation loop
while(!Finished){
  // Determination of the dynamic operating mode, sendings of data
       dependencies and blocking data receptions in sync mode
  ...
  // Local GPU computation
  // Data transfers from node RAM to GPU, GPU grid definition and init
       of shared mem
  CHECK_CUDA_SUCCESS(cudaMemcpyToSymbol(dataOnGPU, dataInRAM,
      inputsSize, 0, cudaMemcpyHostToDevice), "Data␣transfer");
  ...
  // Kernel call
  gpuKernelName<<<Dg,Db>>>(... @\emph{kernel parameters}@ ...);
  // Potential pre/post-treatments in pipeline like computations
  ...
  // Waiting for kernel completion
  cudaDeviceSynchronize();
  // Results transfer from GPU to node RAM
  omp_set_lock(&lockWrite); // Wait for write access to resultsInRam
  CHECK_CUDA_SUCCESS(cudaMemcpyFromSymbol(resultsInRam, resultsOnGPU,
      resultsSize, 0, cudaMemcpyDeviceToHost), "Results␣transfer");
  // Potential post-treatments in non-pipeline computations
  ...
  omp_unset_lock(&lockWrite); // Give back read access to aux thread
  omp_test_lock(&lockRes);
  omp_unset_lock(&lockRes);   // Informs aux thread of new results

  // Auxiliary computations availability checking
  if(omp_test_lock(&lockAux)){
    // Use auxRes to update the iterative process
    ... // May induce additional GPU transfers
  }

  // Convergences checking
  if(curMode == SYNC){
    // Local convergence checking and global exchange of local states
    ...
    // Determination of global convergence (all nodes in local CV)
    if(cvLocale == 1 && nbCVLocales == nbP-1){
      // Stop of the iterative process and sending of end messages
      ...
      // Unlocking of aux thread for termination
      omp_test_lock(&lockRes);
      omp_unset_lock(&lockRes);
    }else{
      // Re-initialization of state information and iteration barrier
      ...
    }
  }
}
```

Listing 7.16: Auxiliary computing function in the final asynchronous scheme with CPU/GPU overlap

```
   // Variables declarations and initialization
   ... auxInput ... // Local array for input data

   // Computation loop
 5 while(!Finished){
     // Data copy from resultsInRam into auxInput
     omp_set_lock(&lockRes);    // Waiting for new results from main comps
     if(!Finished){
       omp_set_lock(&lockWrite); // Waiting for access to results
10     for(ind=0; ind<resultsSize; ++ind){
         auxInput[ind] = resultsInRam[ind];
       }
       omp_unset_lock(&lockWrite); // Give back write access to main
           thread
       // Auxiliary computations with possible interruption at the end
15     for(ind=0; ind<auxSize && !Finished; ++ind){
         // Computation of auxRes array according to auxInput
         ...
       }
       // Informs main thread that new aux results are available in
           auxData
20     omp_test_lock(&lockAux); // Ensures mutex is locked when unlocking
       omp_unset_lock(&lockAux);
     }
   }
```

As can be seen in Listing 7.14, there are three additional mutex (`lockAux`, `lockRes` and `lockWrite`) that are used respectively to inform the main computation thread that new auxiliary results are available (lines 20-21 in Listing 7.16 and line 29 in Listing 7.15), to inform the auxiliary thread that new results from the main thread are available (lines 25-26 in Listing 7.15 and line 7 in Listing 7.16), and to perform exclusive accesses to the results from those two threads (lines 20, 24 in Listing 7.15 and 9, 13 in Listing 7.16). Also, an additional array (`auxRes`) is required to store the results of the auxiliary computations as well as a local array for the input of the auxiliary function (`auxInput`). That last function has the same general organization as the send/receive ones, that is a global loop conditioned by the end of the global process. At each iteration in this function, the thread waits for the availability of new results produced by the main computation thread. This avoids to perform the same computations several times with the same input data. Then, input data of auxiliary computations is copied with a mutual exclusion mechanism. Finally, auxiliary computations are performed. When they are completed, the associated mutex is unlocked to signal the availability of those auxiliary results to the main computing thread. The main thread regularly checks this availability at the end of its iterations and takes them into account whenever this is possible.

Finally, we obtain an algorithmic scheme allowing maximal overlap between CPU and GPU computations as well as communications. It is worth noticing that such scheme is also usable for systems without GPUs but 4-cores nodes.

7.3.4 Experimental validation

As in Section 7.2, we validate the feasibility of our asynchronous scheme with some experiments performed with a representative example of scientific application. It is a three-dimensional version of the advection-diffusion-reaction process that models the evolution of the concentrations of two chemical species in shallow waters. As this process is dynamic in time, the simulation is performed for a given number of consecutive time steps. This implies two nested loops in the iterative process, the outer one for the time steps and the inner one for solving the problem at each time. Full details about this PDE problem can be found in [20]. That two-stage iterative process implies a few adaptations of the general scheme presented above in order to include the outer iterations over the time steps, but the inner iterative process closely follows the same scheme.

We show two series of experiments performed with 16 nodes of the first cluster described in Section 7.2.5. The first one deals with the comparison of synchronous and asynchronous computations. The second one is related to the use of auxiliary computations. In the context of our PDE application, they consist in the update of the Jacobian of the system.

Synchronous and asynchronous computations

The first experiment allows us to check that the asynchronous behavior obtained with our scheme corresponds to the expected one according to its synchronous counterpart. So, we show in Figure 7.6 the computation times of our test application in both modes for different problem sizes. The size shown is the number of discrete spatial elements on each side of the cube representing the 3D volume. Moreover, for each of these elements, there are the concentrations of the two chemical species considered. So, for example, size 30 corresponds in fact to $30 \times 30 \times 30 \times 2$ values.

The results obtained show that the asynchronous version is sensibly faster than the synchronous one for smaller problem sizes, then it becomes similar or even a bit slower for larger problem sizes. A closer comparison of computation and communication times in each execution confirms that this behavior is consistent. The asynchronous version is interesting if communication time is similar or larger than computation time. In our example, this is the case up to a problem size between 50 and 60. Then, computations become longer than communications. Since asynchronous computations often require more iterations to converge, the gain obtained on the communication side becomes smaller than the overhead generated on the computation side, and the asynchronous version takes longer.

Overlap of auxiliary computations

In this experiment, we use only the asynchronous version of the application. In the context of our test application, we have an iterative PDE solver based on

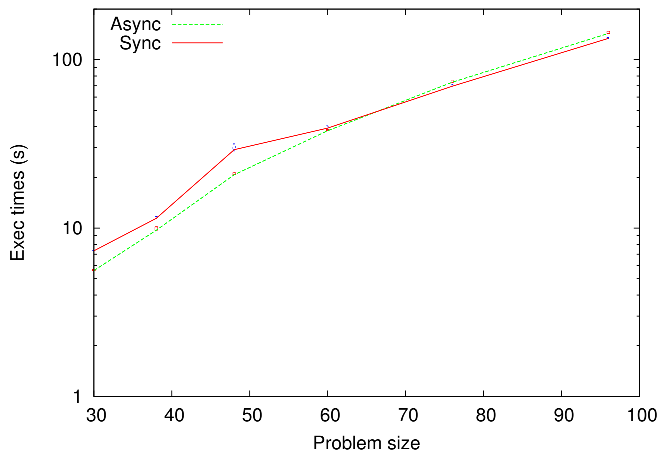

FIGURE 7.6: Computation times of the test application in synchronous and asynchronous modes.

Netwon resolution. Such solvers are written under the form $x = T(x),\ x \in \mathbb{R}^n$ where $T(x) = x - F'(x)^{-1}F(x)$ and F' is the Jacobian of the system. In such cases, it is necessary to compute the vector Δx in $F' \times \Delta x = -F$ to update x with Δx. There are two levels of iterations, the inner level to get a stabilized version of x, and the outer level to compute x at the successive time steps in the simulation process. In this context, classical algorithms either compute F' only at the first iteration of each time step or at some iterations but not all because the computation of F' is done in the main iterative process and it has a relatively high computing cost.

However, with the scheme presented above, it is possible to continuously compute new versions of F' in parallel to the main iterative process without penalizing it. Hence, F' is updated as often as possible and taken into account in the main computations when it is relevant. So, the Newton process should be accelerated a little bit.

We compare the performance obtained with overlapped Jacobian updatings and non-overlapped ones for several problem sizes, see Figure 7.7.

The overlap is clearly efficient as the computation times with overlapping Jacobian updatings are much better than without overlap. Moreover, the ratio between the two versions tend to increase with the problem size, which is as expected. Also, we have tested the application without auxiliary computations at all, that is, the Jacobian is computed only once at the beginning of each time step of the simulation. The results for this last version are quite similar to the overlapped auxiliary computations, and even better for small problem sizes. The fact that no sensible gain can be seen on this range of problem sizes is due to the limited number of Jacobian updates taken into account in the main computation. This happens when the Jacobian update is as long

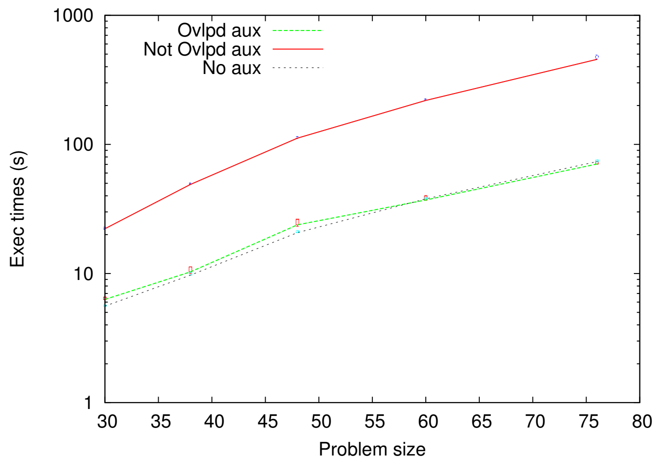

FIGURE 7.7: Computation times with or without overlap of Jacobian updatings in asynchronous mode.

as several iterations of the main process. So, the benefit is reduced in this particular case.

Those results show two things; first, auxiliary computations do not induce great overhead in the whole process. Second, for this particular application the choice of updating the Jacobian matrix as auxiliary computations does not speed up the iterative process. This does not question the parallel scheme in itself but merely points out the difficulty to identify relevant auxiliary computations. Indeed, this identification depends on the considered application and requires a profound specialized analysis.

Another interesting choice could be the computation of load estimation for dynamic load balancing, especially in decentralized diffusion strategies where loads are transferred between neighboring nodes [8]. In such case, the load evaluation and the comparison with other nodes can be done in parallel of the main computations without perturbing them.

7.4 Perspective: A unifying programming model

In the previous sections we have seen that controlling a distributed GPU application when using tools that are commonly available is a quite challenging task. To summarize, such an application has components that can be roughly classified as:

CPU: CPU bound computations, realized as procedures in the chosen programming language

CUDA$_{kern}$: GPU bound computations, in our context realized as CUDA compute kernels

CUDA$_{trans}$: data transfer between CPU and GPU, realized with CUDA function calls

MPI: distributed data transfer routines, realized with MPI communication primitives

OpenMP: inter-thread control, realized with OpenMP synchronization tools such as mutexes

CUDA$_{sync}$ synchronization of the GPU, realized with CUDA functions

Among these, the last (CUDA$_{sync}$) is not strictly necessary on modern systems, but still recommended to obtain optimal performance. With or without that last step, such an application is highly complex: it is difficult to design or to maintain, and depends on a lot of different software components. The goal of this section is to present a new path of development that allows to replace the last three or four types of components that control the application (MPI, OpenMP, CUDA$_{sync}$ and eventually CUDA$_{trans}$) by a single tool: **O**rdered **R**ead-**W**rite **L**ocks, ORWL, see [10, 13]. Besides the simplification of the algorithmic scheme that we already have mentioned, the ongoing implementation of ORWL allows to use a feature of modern platforms that can improve the performance of CPU bound computations: lock-free atomic operations to update shared data consistently. For these, ORWL relies on new interfaces that are available with the latest revision of the ISO standard for the C programming language, see [16].

7.4.1 Resources

ORWL places all its concepts that concern data and control around a single abstraction: *resources*. An ORWL resource may correspond to a local or remote entity and is identified through a *location*, that is a unique identification through which it can be accessed from all different components of the same application. In fact, resources and locations (entities and their names so to speak) are mostly identified by ORWL and these words will be used interchangeably.

Resources may be of very different kind:

Data resources are entities that represents data and not specific memory buffers or locations. During the execution of an application it can be *mapped* repeatedly into the address space and in effect be represented at different addresses. Data resources can be made accessible uniformly in all parts of the application, provided that the locking protocol is observed, see below. Data resources can have different persistence:

- **RAM** data resources are typically temporary data that serve only during a single run of the application. They must be initialized at the beginning of their lifetime and the contents is lost at the end.
- **File** data resources are persistent and linked to a file in the file system of the platform.
- **Collective** data resources are data to which all tasks of an application contribute (see below). Examples for such resources are *broadcast*, *gather* or *reduce* resources, e.g to distribute initial data or to collect the result of a distributed computation.

Other types of data resources could be easily implemented with ORWL, e.g web resources (through a ftp, http or whatever server address) or fixed hardware addresses.

Device resources represent hardware entities of the platform. ORWL can then be used to regulate the access to such device resources. In our context the most important such resource is the GPU, but we could easily use it to represent a CPU core, a camera or other peripheral device.

Listing 7.17 shows an example of a declaration of four resources per task. Two (`curBlock` and `nextBlock`) are intended to represent the data in a block-cyclic parallel matrix multiplication (see p. 120), `GPU` represents a GPU device and `result` will represent a collective "gather" resource among all the tasks.

Listing 7.17: Declaration of ORWL resources for a block-cyclic matrix multiplication

```
#include "orwl.h"
...
ORWL_LOCATIONS_PER_TASK(curBlock, nextBlock, GPU, result);
ORWL_DATA_LOCATION(curBlock);
ORWL_DATA_LOCATION(nexBlock);
ORWL_DEVICE_LOCATION(GPU);
ORWL_GATHER_LOCATION(result);
```

7.4.2 Control

ORWL regulates access to all its resources, no "random access" to a resource is possible. It doesn't even have a user-visible data type for resources.

- All access is provided through *handle*s. Similar to pointers or links, these only refer to a resource and help to manipulate it. Usually several handles to the same resource exist, even inside the same OS process or thread, or in the same application task.
- The access is locked with RW semantics, where *R* stands for concurrent **R**ead access, and *W* for exclusive **W**rite access. This feature replaces the control aspect of MPI communications, OpenMP inter-thread control and CUDA$_{sync}$.
- This access is **O**rdered (or serialized) through a FIFO, *one FIFO per resource*. This helps to run the different tasks of an application in a controlled order and to always have all resources in a known state. This aspect largely replaces and extends the ordering of tasks that MPI typically achieves through the passing of messages.
- The access is transparently managed for remote or local resources. Communication, if necessary, is done asynchronously behind the scenes. This replaces the explicit handling of buffers and messages with MPI.

7.4.3 Example: block-cyclic matrix multiplication (MM)

Let us now have a look how a block-cyclic matrix multiplication algorithm can be implemented with these concepts (Listing 7.18). Inside the loop it mainly consists of three different operations, of which the first two can be run concurrently, and the third must be done after the other two.

Listing 7.18: Block-cyclic matrix multiplication, high level per task view

```
typedef double MBlock[N][N];
MBlock A;
MBlock B[k];
MBlock C[k];

<do some initialization>

for (size_t i = 0; i < k; ++i) {
  MBlock next;
  parallel-do {
    operation 1: <copy the matrix A of the left neighbor into next>;
    operation 2: {
      <copy the local matrix A to the GPU >;
      <on GPU perform C[i] = A * B[0] + ... + A * B[k-1]; >;
    }
  }
  operation 3: {
    <wait until the right neighbor has read our block A>;
    A = next;
  }
}

<collect the result matrix C consisting of all C blocks>
```

Listing 7.19 shows the local copy operation 3 as it could be realized with ORWL. It uses two resource handles `nextRead` and `curWrite` and marks nested *critical sections* for these handles. Inside the nested sections it obtains pointers to the resource data; the resource is *mapped* into the address space of the program, and then a standard call to `memcpy` achieves the operation itself. The operation is integrated in its own **for**-loop, such that it could run independently in an OS thread by its own.

Listing 7.19: An iterative local copy operation

```
for (size_t i = 0; i < k; ++i) {
  ORWL_SECTION(nextRead) {
    MBlock const* sBlock = orwl_read_map(nextRead);
    ORWL_SECTION(curWrite) {
      MBlock * tBlock = orwl_write_map(curWrite);
      memcpy(tBlock, sBlock, sizeof *tBlock);
    }
  }
}
```

Next, in Listing 7.20 we copy data from a remote task to a local task. Substantially the operation is the same, only that in the example different handles (`remRead` and `nextWrite`) are used to represent the respective resources.

Listing 7.20: An iterative remote copy operation as part of a block cyclic matrix multiplication task

```
for (size_t i = 0; i < k; ++i) {
  ORWL_SECTION(remRead) {
    MBlock const* sBlock = orwl_read_map(remRead);
    ORWL_SECTION(nextWrite) {
      MBlock * tBlock = orwl_write_map(nextWrite);
      memcpy(tBlock, sBlock, sizeof *tBlock);
    }
  }
}
```

Now let us have a look into the operation that probably interests us the most, the interaction with the GPU in Listing 7.21. Again there is much structural resemblance to the copy operations from above, only that we transfer the data to the GPU in the innermost block and then run the GPU MM kernel while we still are inside the critical section for the GPU.

Listing 7.21: An iterative GPU transfer and compute operation as part of a block cyclic matrix multiplication task

```
for (size_t i = 0; i < k; ++i) {
  ORWL_SECTION(GPUhandle) {
    ORWL_SECTION(curRead) {
      MBlock const* sBlock = orwl_read_map(curRead);
      transferToGPU(sBlock, i);
    }
    runMMonGPU(i);
  }
}
```

Now that we have seen how the actual procedural access to the resources is regulated we will show how the association between handles and resources is specified. E.g in our application of block-cyclic MM the `curRead` handle should correspond to current matrix block of the corresponding task, whereas remRead should point to the current block of the neighboring task. Both read operations on these matrix blocks can be effected without creating conflicts, so we would like to express that fact in our resource specification. From a point of view of the resource "current block" of a particular task, this means that it can have two simultaneous readers, the task itself performing the transfer to the GPU, and the neighboring task transferring the data to its "next block".

Listing 7.22 first shows the local dynamic declarations of our application; it declares a block type for the matrix blocks, a result data for the collective resource, and the six handles that we have seen so far.

Listing 7.22: Dynamic declaration of handles to represent the resources

```
/* A type for the matrix blocks */
typedef double MBlock[N][N];
/* Declaration to handle the collective resource */
ORWL_GATHER_DECLARE(MBlock, result);

/* Variables to handle data resources */
orwl_handle2 remRead   = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 nextWrite = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 nextRead  = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 curWrite  = ORWL_HANDLE2_INITIALIZER;
orwl_handle2 curRead   = ORWL_HANDLE2_INITIALIZER;

/* Variable to handle the device resources */
orwl_handle2 GPUhandle = ORWL_HANDLE2_INITIALIZER;
```

With these declarations, we didn't yet tell ORWL much about the resources to which these handles refer, nor the type (read or write) or the priority (FIFO position) of the access. This is done in code Listing 7.23. The handles for Listing 7.21 are given first, GPUhandle will be accessed exclusively (therefore the write) and, as said, curRead is used shared (so a read). Both are inserted in the FIFO of there respective resources with highest priority, specified by the 0s in the third function parameter. The resources to which they correspond are specified through calls to the macro ORWL_LOCATION, indicating the task (orwl_mytid is the ID of the current task) and the specific resource of that task, here GPU and curBlock.

Likewise, a second block of insertions concerns the handles for Listing 7.20: newWrite reclaims an exclusive access and remRead a shared. remRead corresponds to a resource of another task; previous(orwl_mytid) is supposed to return the ID of the previous task in the cycle. Both accesses can be effected concurrently with the previous operation, so we insert with the same priority 0 as before.

Then, for the specification of the third operation (Listing 7.19) we need to use a different priority: the copy operation from nextBlock to curBlock has to be performed after the other operations have terminated.

As a final step, we then tell ORWL that the specification of all accesses is complete and that it may schedule all these accesses in the respective FIFOs of the resources.

Listing 7.23: Dynamic initialization of access mode and priorities

```
/* One operation with priority 0 (highest) consists      */
/* in copying from current to the GPU and run MM, there. */
orwl_write_insert(&GPUhandle, ORWL_LOCATION(orwl_mytid, GPU), 0);
orwl_read_insert(&curRead, ORWL_LOCATION(orwl_mytid, curBlock), 0);

/* Another operation with priority 0 consists            */
/* in copying from remote to next                        */
orwl_read_insert(&remRead, ORWL_LOCATION(previous(orwl_mytid),
    curBlock), 0);
orwl_write_insert(&nextWrite, ORWL_LOCATION(orwl_mytid, nextBlock), 0)
    ;

/* One operation with priority 1 consists       */
/* in copying from next to current              */
orwl_read_insert(&nextRead, ORWL_LOCATION(orwl_mytid, nextBlock), 1);
orwl_write_insert(&curWrite, ORWL_LOCATION(orwl_mytid, curBlock), 1);

orwl_schedule();
```

7.4.4 Tasks and operations

With that example we have now seen that ORWL distinguishes *tasks* and *operations*. An ORWL program is divided into tasks that can be seen as the algorithmic units that will concurrently access the resources that the program uses. A task for ORWL is characterized by

- a fixed set of resources that it manages, "*owns*", in our example the four resources that are declared in Listing 7.17.
- a larger set of resources that it accesses, in our example all resources that are used in Listing 7.23.
- a set of operations that act on these resources, in our example the three operations that are used in Listing 7.18, and that are elaborated in Listings 7.19, 7.20 and 7.21.

Each ORWL operation is characterized by

- one resource, usually one that is owned by the enclosing task, that it accesses exclusively. In our example, operation 1 has exclusive access to the `next` block, operation 2 has exclusive access the `GPU` resource, and operation 3 to the `A` block.
- several resources that are accessed concurrently with others.

In fact each ORWL operation can be viewed as a compute-and-update procedure of a particular resource with knowledge of another set of resources.

7.5 Conclusion

In this chapter, different methodologies that effectively take advantage of current cluster technologies with GPUs have been presented. Beyond the simple collaboration of several nodes that include GPUs, we have addressed parallel schemes to efficiently overlap communications with computations (including GPU transfers), and also computations on the CPU with computations on the GPU. Moreover, parallel schemes for synchronous as well as asynchronous iterative processes have been proposed. The proposed schemes have been validated experimentally and provide the expected behavior. Finally, as a prospect we have developed a programming tool that will, in middle or long term, provide all the required technical elements to implement the proposed schemes in a single tool, without requiring several external libraries.

We conclude that GPU computations are very well suited to achieve overlap with CPU computations and communications and they can be fully integrated in algorithmic schemes combining several levels of parallelism.

7.6 Glossary

AIAC: Asynchronous Iterations - Asynchronous Communications.

Asynchronous iterations: Iterative process where each element is updated without waiting for the last updates of the other elements.

Auxiliary computations: Optional computations performed in parallel to the main computations and used to complete them or speed them up.

BSP parallel scheme: Bulk Synchronous Parallel, a parallel model that uses a repeated pattern (superstep) composed of: computation, communication, barrier.

GPU stream: Serialized data transfers and computations performed on a same piece of data.

Message loss/miss: Can be said about a message that is either not sent or sent but not received (possible with unreliable communication protocols).

Message stamping: Inclusion of a specific value in messages of the same tag to distinguish them (kind of secondary tag).

ORWL: Ordered Read Write Locks, a programming tool proposing a unified programming model.

Page-locked data: Data that are locked in cache memory to ensure fast accesses.

Residual: Difference between results of consecutive iterations in an iterative process.

Streamed GPU sequence: GPU transfers and computations performed simultaneously via distinct GPU streams.

Bibliography

[1] Message passing interface. `http://www.mpi-forum.org/docs`.

[2] OpenMP multi-threaded programming API. `http://www.openmp.org`.

[3] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Asynchronism for iterative algorithms in a global computing environment. In *The 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'2002)*, pages 90–97, Moncton, Canada, June 2002.

[4] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing*, 31(5):439–461, 2005.

[5] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Performance comparison of parallel programming environments for implementing AIAC algorithms. *Journal of Supercomputing. Special Issue on Performance Modelling and Evaluation of Parallel and Distributed Systems*, 35(3):227–244, 2006.

[6] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. *Parallel Iterative Algorithms: from sequential to grid computing.* Numerical Analysis & Scientific Computing Series. Chapman & Hall/CRC, 2007.

[7] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. An efficient and robust decentralized algorithm for detecting the global convergence in asynchronous iterative algorithms. In *8th International Meeting on High Performance Computing for Computational Science, VECPAR'08*, pages 251–264, Toulouse, June 2008.

[8] Jacques M. Bahi, Sylvain Contassot-Vivier, and Arnaud Giersch. Load balancing in dynamic networks by bounded delays asynchronous diffusion. In J.M.L.M. Palma et al., editor, *VECPAR 2010*, volume 6449 of *LNCS*, pages 352–365. Springer, Heidelberg, 2011. `DOI:~10.1007/978-3-642-19328-6\33`.

[9] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice Hall, Englewood Cliffs, New Jersey, 1999.

[10] Pierre-Nicolas Clauss and Jens Gustedt. Iterative Computations with Ordered Read-Write Locks. *Journal of Parallel and Distributed Computing*, 70(5):496–504, 2010.

[11] Sylvain Contassot-Vivier, Thomas Jost, and Stéphane Vialle. Impact of asynchronism on GPU accelerated parallel iterative computations. In *PARA 2010 conference: State of the Art in Scientific and Parallel Computing*, Reykjavík, Iceland, June 2010.

[12] A. Frommer and D. B. Szyld. On asynchronous iterations. *J. Comput. and Appl. Math.*, 123:201–216, 2000.

[13] Jens Gustedt and Emmanuel Jeanvoine. Relaxed Synchronization with Ordered Read-Write Locks. In Michael Alexander et al., editors, *Euro-Par 2011: Parallel Processing Workshops*, volume 7155 of *LNCS*, pages 387–397, Bordeaux, France, May 2012. Springer.

[14] Jens Gustedt, Stéphane Vialle, and Amelia De Vivo. The parXXL environment: Scalable fine grained development for large coarse grained platforms. In Bo Kågström et al., editors, *PARA 06*, volume 4699, pages 1094–1104, Umeå, Sweden, 2007. Springer.

[15] Torsten Hoefler and Andrew Lumsdaine. Overlapping communication and computation with high level communication routines. *Cluster Computing and the Grid, IEEE International Symposium on*, pages 572–577, 2008. `http://doi.ieeecomputersociety.org/10.1109/CCGRID.2008.15`.

[16] JTC1/SC22/WG14, editor. *Programming languages - C.* Number ISO/IEC 9899. ISO, cor. 1:2012 edition, 2011.

[17] NVIDIA. *NVIDIA CUDA C Best Practices Guide 4.0*, May 2011.

[18] NVIDIA. *NVIDIA CUDA C Programming Guide 4.0*, June 2011.

[19] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[20] S. Vialle, S. Contassot-Vivier, and T. Jost. *Handbook of Energy-Aware and Green Computing*, chapter Optimizing Computing and Energy Performances in Heterogeneous Clusters of CPUs and GPUs. Computer & Information Science Series. Chapman and Hall/CRC, Jan 2012.

[21] Stéphane Vialle and Sylvain Contassot-Vivier. *Patterns for parallel programming on GPUs*, chapter Optimization methodology for Parallel Programming of Homogeneous or Hybrid Clusters. Saxe-Coburg Publications, February 2013. to appear.